CAPTCHA SOLVER

Matteo Mariotti - 1936936

Marco Realacci - 1938880

Antonio Pietro Romito - 1932500

Federico Pizzari - 1936451

INDEX

INTRODUCTION

After intensive research of our topic, we have at last decided to try to develop a neural network that could resolve captchas.

But what's a captcha?
A Captcha (Completely Automated Public Turing test to tell Computers and Humans Apart) is a type of challenge that aims to determine whether a service user is really a human. The type of Captcha that we have worked on is an image that contains some weirdly written characters and some "visual fuzz" and we are required to correctly guess the characters inside it in order to proceed with the execution of our program.

Nowadays, computers have become increasingly better at deciphering these types of images and now these images are so difficult to decode that even humans are struggling to solve them. In fact, some sites don't even use image captchas anymore, but they prefer to use some other types of automatic checks that can fall back to a more standard captcha if they fail.

But some older sites still use this type of captchas and being able to solve them it's still a complicated task that could be challenging for a machine learning algorithm.

OUR APPROACH

We started our research by heading on Kaggle, a machine learning centered website full of great datasets, attempting to find one for captchas.
We found two great datasets with lots of entries at the following links:

- **DATASET A:** The first one has images that are somewhat distorted and have a line passing through their character. LINK

- **DATASET B:** The second one has colored characters placed in random places. Furthermore, we have both lowercase and uppercase characters. LINK

The idea behind our implementation

Our first idea was to “clean a little” our captcha images and then train our neural network directly with those, but we realized pretty soon that it wasn’t really as good as we thought. In fact, in this way we lose a lot of potential because if we train our network using one character at a time, we provide a much bigger dataset that very often translates to a much more precise guess.

So we decided to create two new datasets to train the models, created by manipulating the images using OpenCV and extracting the characters individually from the captchas.

The final architecture

The graph on the next page shows the architecture of the system.

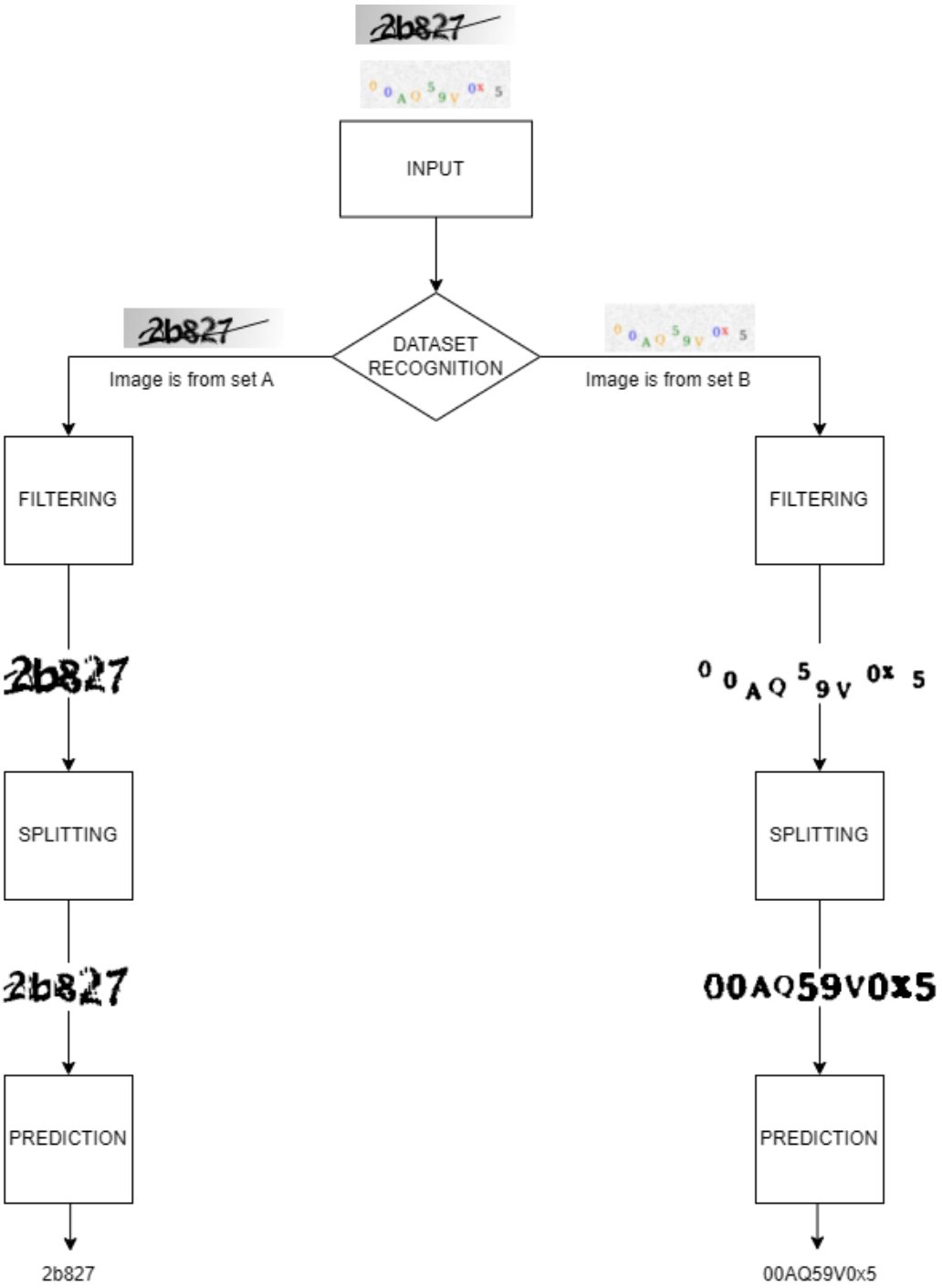
We first recognize if the CAPTCHA is from dataset A or dataset B. We can then proceed to process the images to improve the readability of the characters and to output a binary image. Then we try to identify the characters and split them into different images (an image per each character).

Finally, we use our models to predict the value of each character.

Why two models?

We decided to build two different models for each dataset, as there are a lot of differences between them, so we can train each model specifically for a type of character.

Also, characters in dataset A are bigger in size (but also harder to read) than in dataset B, thus requiring a bigger neural network.



the architecture of the system

BUILDING THE DATASETS

We wrote two Python scripts to process the images from the original datasets to produce ours, which are made of black and white characters.

We went for binary images as there's no need to know the color (or the gradient) of a character, it would have only slowed down the training phase of the models.

PROCESSING Dataset A

Filtering

Initially, we used some OpenCV filters to remove the background and improve the readability of the characters, and to help the splitting algorithm to better distinguish each one. We used the following filters in this order:

1 - Thresholding

To remove the background noise in the images, we used OpenCV's threshold filter, with a threshold of 127 and THRESH_BINARY as the threshold type.

2 - Morphological closing

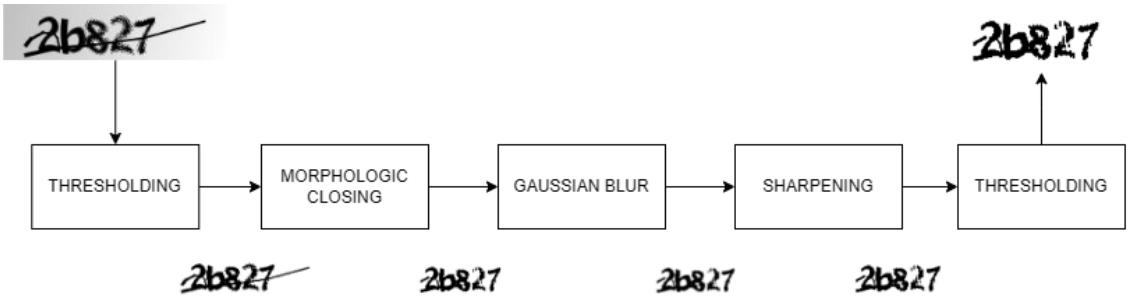
We used Morphological Closing in an attempt to isolate each character from each other. The closing operation is performed by first applying a morphological dilation to the binary image using a kernel (3x1 in our case), followed by an erosion using the same kernel. This can help to open the gaps in the foreground objects of the binary image. In our case, it massively helped to remove the line that goes through the characters. We obviously chose a kernel that wasn't too harsh on the image, as this could corrupt in an excessive way our characters. We tried to use other operators to do this kind of job, but this transformation was the one that gave the better results.

3 - Gaussian blur + sharpening

To smooth the image and remove some of the noise. The combination of blurring + sharpening helps in giving out a clearer image, with fewer artifacts.

4 - Thresholding

To have a binary image, we used the binary thresholding filter again, with a threshold level of 185. In this case, we were able to use such a high value because at this point the characters in the image correspond to the parts with a more "distinct" color.



Character splitting

Once we had some much clearer images, we proceeded to identify the characters and save them as separate images.

We first used the MSER algorithm to determine regions in the image, followed by some filtering to merge and split the regions:

- if a region is too large, it will most likely contain two or more characters. Based on the width of the region, we split it from 1 (no splitting) to 5 different equal subregions
- if a region contains 99% of white pixels, it certainly will not contain a character, so we can safely ignore it
- there aren't characters above other characters, so if a region is above another region, they will refer to the same character: we will merge them into a single region
- if two regions overlap and their edges are very close to each other, they

probably refer to the same character, so we merge them into a single region
- if there are still very small regions (less than 20px in height and less than 10px in width), we can ignore them, as they will certainly not contain a character.

We have found that 87% of the time, after all the processing, we are left with 5 regions: in this case, we can be very confident that they do match each character.

If the number of regions is not 5, they will be discarded, and the image will be split manually using fixed-position regions.

This approach is possible because the characters are often placed roughly in the same position in all images, but it's obviously a suboptimal approach. In fact, when we tried to train our model using a dataset generated only by using the "fixed-position" approach, our model final accuracy decreased by 4-5%.

Data augmentation

As the dataset is not that big (5350 images of characters), we decided to use a technique called "data augmentation" to increase the number of images in the set.

We did it by rotating each image by a random angle between -45° and 45°.

This way, we doubled the size of the dataset and we improved the accuracy in the cases where the images are "naturally" slightly rotated in the CAPTCHA

PROCESSING Dataset B

In the dataset B, characters are clearer and easier to isolate from each other, the images are in RGB format, which means that some characters are colored. Also, there is some noise in the background.

Filtering

The background noise is made by gray pixels, which are never darker than (200, 200, 200). At the same time, gray characters are never that bright, so we can easily filter out the noise by making white every pixel lighter than (200, 200, 200).

To remove artifacts within the characters, we used the same technique as before: blurring and then sharpening the image.

At this point, the image is clear enough that we can convert it to a binary image, by using OpenCV's thresholding: with a threshold of 220 and THRESH_BINARY as the thresholding type.

Character splitting

As before, we used the MSER algorithm to find regions in the CAPTCHA images.

Then we removed regions that are inside other regions: this helps in cases like the one shown below:

SAVING THE DATA

All the characters are saved in a different directory for each dataset, and each file is named with an incrementing number.

The model needs to know which character represents the image to be trained, so we need to also store a label for each image. To achieve this, the Python script also saves a .csv file that associates the name of each file with the label that the image represents. That is possible due to the fact that every image filename represents the content of the captcha, so the script is able to associate every found character with the corresponding letter.

PREPARING THE DATA

We then had to prepare the datasets in a way that is suitable for training the model. For that, we prepared each model creating a class called CaptchaDataset, which inherits from the torch.utils.data.Dataset class. This class has two Python's dunder methods: **__init__** and **__getitem__**.

- The __init__ method (the constructor) is used to initialize the dataset.

- The __getitem__ method is used to retrieve a sample from the dataset. It takes an index as an argument and returns a tuple containing the image and the label. The returned image is a PyTorch tensor representing the binary image.

With the help of the torch.utils.data.random_split function, we split the dataset in two: the **training set** and the **testing set**, with an 80/20 split ratio. This means that 80% of the dataset will be used for training the model, while the remaining 20% will be used for testing it.

At this point, we were ready to define the models.

DEFINING THE MODELS

To define the models, we defined a class called ReteNeurale which inherits from the torch.nn.Module class. Then, from the constructor of the class, we defined the layers of the neural network.

The model for dataset A

Initially, we started with a network made of linear (fully connected) layers only, but it wasn't performing really well: in the tests, we achieved a maximum accuracy of 40%.

In the end, we opted for a **convolutional neural network**, as they are great for finding patterns in images to recognize objects (characters in our case).

The CNN is composed of three convolutional layers, followed by three fully connected layers. The convolutional layers are responsible for extracting features from the input image, such as edges, shapes, and patterns. The fully connected layers are responsible for classifying the extracted features into one of the 36 possible characters (26 letters and 10 digits).

The **first convolutional layer** (conv1) takes a single-channel image as an input and applies a 4x4 kernel to it. Each kernel produces a feature map that represents the presence of a certain feature in the input image. The output of this layer is a tensor of shape (20, 47, 47), where 20 is the number of channels and 47 is the reduced height and width of the feature maps after applying the filters.

The **second convolutional layer** (conv2) takes the output of the first convolutional layer as an input and applies a kernel of size 4x4 with a stride of 2 to it. The stride means that the filters move two pixels at a time, resulting in a smaller output tensor. The output of this layer is a tensor of shape (32, 22, 22), where 32 is the number of channels and 22 is the reduced height and width of the feature maps after applying the filters.

The **third convolutional layer** (conv3) takes the output of the second convolutional layer as an input and applies a kernel of size 4x4 with a stride of 2 to it. The output of this layer is a tensor of shape (64, 10, 10), where 64 is the number of channels and 10 is the reduced height and width of the feature maps after applying the filters.

After each convolutional layer, a **batch normalization layer** (batchNorm1, batchNorm2, batchNorm3) is applied to normalize the output tensors and improve the stability and performance of the network. Batch normalization reduces the internal covariate shift, which is the change in the distribution of inputs to each layer due to the updates of previous layers. Using batch normalization we can reduce the time needed to train the network and we are able to use higher learning rates.

After the third convolutional layer, a **max pooling layer** (pool) is applied to reduce the size of the output tensor and introduce some spatial invariance. Max pooling selects the maximum value in each 2x2 region of the feature map and discards the rest. The output of this layer is a tensor of shape (64, 5, 5), where 64 is the number of filters and 5 is the reduced height and width of the

feature maps after applying max pooling.

The output tensor of the max pooling layer is then flattened into a vector of length 1600 (x.view(-1, 1600)) and fed into the **first fully connected layer** (fc1). This layer has 512 neurons and applies a linear transformation to the input vector. The output of this layer is a vector of length 512.

The **second fully connected layer** (fc2) takes the output vector of length 512 as an input and applies another linear transformation to it. The output of this layer is a vector of length 256.

The **third fully connected layer** (fc3) takes the output vector of length 256 as an input and applies another linear transformation to it. The output of this layer is a vector of length 36, which represents the logits (unnormalized probabilities) for each possible character class.

The model for dataset B

The model for the dataset B has the same structure as the other model, but different layer dimensions. That is because the images in the set are smaller, so we had to change the dimensions of the layers in order to have a workable size at the end of the convolutional layers.

All the convolutional layers have a 3x3 kernel, instead of 4x4. The first layer has 16 outputs, the second layer has 32 outputs, and the third layer has 64 outputs.

The linear layers are only two, the first one has 256 inputs and outputs, and the second one has 62 outputs, to represent all the possible characters.

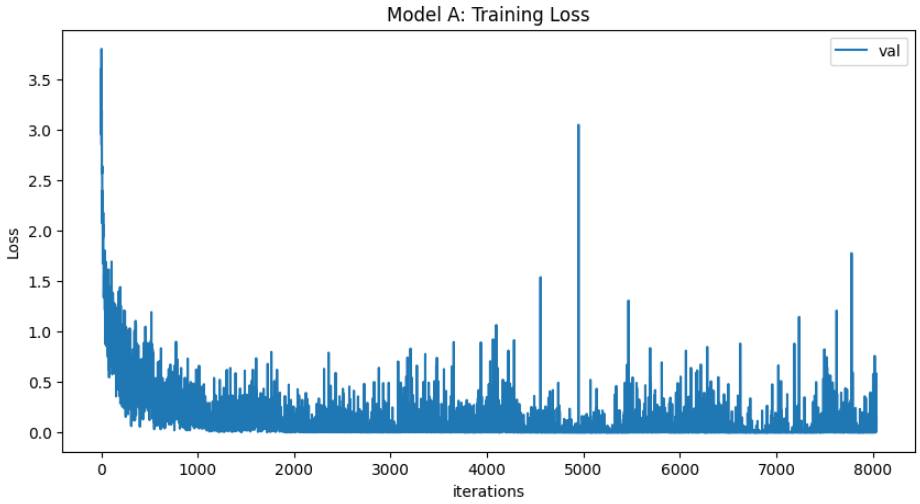
TRAINING THE MODELS

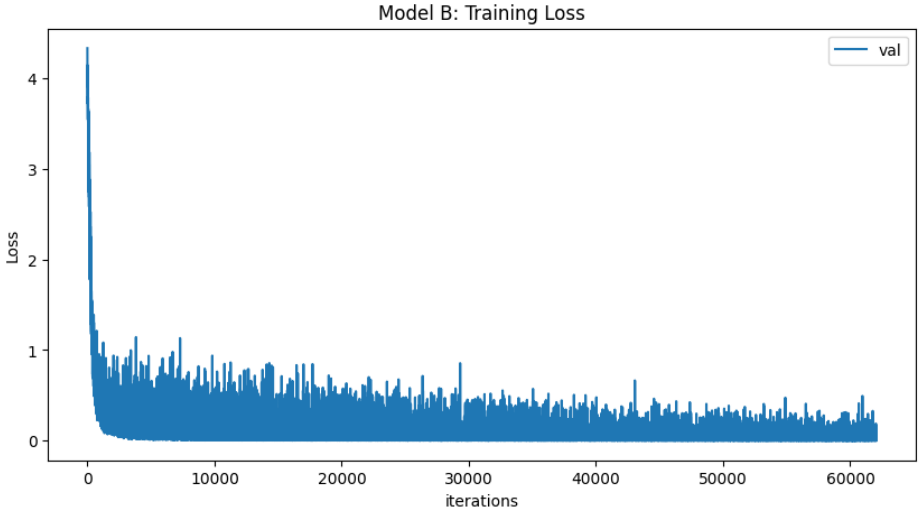
As we said before, we split each dataset into a training and a testing set to train the models. We then trained each model for 30 epochs with a batch size of 32. At the end of each training epoch, we did a validation on the test set to have confirmation that the training process was proceeding as expected.

We collected the training loss while the process was going to plot a graph at the end of the computation.

Training results

For **model A**, we managed to reach an accuracy of **88%** on the testing set, whereas, for **model B** we managed to reach an accuracy of **97%**!





USING THE MODELS

Now we want to use the models to solve real CAPTCHAs.

The model_tester.py program takes as input an image containing a complete captcha (from one of the two starting datasets) and gives as output the predicted value of the text in the captcha.

We first started by recognizing which of the two datasets the image is from, that is an easy task considering the different dimensions of the images. We then proceeded to apply the correct processing algorithms and use the correct pre-trained model to predict each character.

The predicted values are then printed onto the console (standard output).

Execution example

File: samples/type_A/728n8.png

Execution result:

```
Processing file: samples/type_A/728n8.png
image is from SET 1
splitting the image in characters...
loading the model...
predicting (using cuda platform)...
Predicted value: 728n8
~/a/captcha-solver on main ×
```